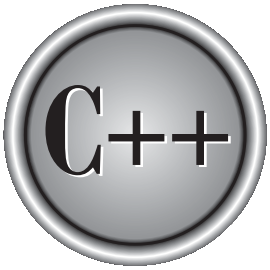
The Complete Reference



Chapter 4

Arrays and Null-Terminated Strings

An *array* is a collection of variables of the same type that are referred to through a common name. A specific element in an array is accessed by an index. In C/C++, all arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. Arrays may have from one to several dimensions. The most common array is the *null-terminated string*, which is simply an array of characters terminated by a null.

Arrays and pointers are closely related; a discussion of one usually refers to the other. This chapter focuses on arrays, while Chapter 5 looks closely at pointers. You should read both to understand fully these important constructs.

Single-Dimension Arrays

The general form for declaring a single-dimension array is

type var_name[size];

Like other variables, arrays must be explicitly declared so that the compiler may allocate space for them in memory. Here, *type* declares the base type of the array, which is the type of each element in the array, and *size* defines how many elements the array will hold. For example, to declare a 100-element array called **balance** of type **double**, use this statement:

```
double balance[100];
```

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example,

```
balance[3] = 12.23;
```

assigns element number 3 in **balance** the value 12.23.

In C/C++, all arrays have 0 as the index of their first element. Therefore, when you write

```
char p[10];
```

you are declaring a character array that has ten elements, **p[0]** through **p[9]**. For example, the following program loads an integer array with the numbers 0 through 99:

```
#include <stdio.h>

int main(void)
{
```

```
  int x[100]; /* this declares a 100-integer array */
  int t;

  /* load x with values 0 through 99 */
  for(t=0; t<100; ++t) x[t] = t;

  /* display contents of x */
  for(t=0; t<100; ++t) printf("%d ", x[t]);

  return 0;
}
```

The amount of storage required to hold an array is directly related to its type and size. For a single-dimension array, the total size in bytes is computed as shown here:

total bytes = sizeof(base type) x size of array

C/C++ has no bounds checking on arrays. You could overwrite either end of an array and write into some other variable's data or even into the program's code. As the programmer, it is your job to provide bounds checking where needed. For example, this code will compile without error, but is incorrect because the **for** loop will cause the array **count** to be overrun.

```
int count[10], i;

/* this causes count to be overrun */
for(i=0; i<100; i++) count[i] = i;
```

Single-dimension arrays are essentially lists of information of the same type that are stored in contiguous memory locations in index order. For example, Figure 4-1 shows how array **a** appears in memory if it starts at memory location 1000 and is declared as shown here:

```
char a[7];
```

Element	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Address	1000	1001	1002	1003	1004	1005	1006

Figure 4-1. *A seven-element character array beginning at location 1000*

Generating a Pointer to an Array

You can generate a pointer to the first element of an array by simply specifying the array name, without any index. For example, given

```
int sample[10];
```

you can generate a pointer to the first element by using the name **sample**. Thus, the following program fragment assigns **p** the address of the first element of **sample**:

```
int *p;
int sample[10];

p = sample;
```

You can also specify the address of the first element of an array using the **&** operator. For example, **sample** and **&sample[0]** both produce the same results. However, in professionally written C/C++ code, you will almost never see **&sample[0]**.

Passing Single-Dimension Arrays to Functions

In C/C++, you cannot pass an entire array as an argument to a function. You can, however, pass to the function a pointer to an array by specifying the array's name without an index. For example, the following program fragment passes the address of **i** to **func1( )**:

```
int main(void)
{
  int i[10];

  func1(i);
  .
  .
  .
}
```

If a function receives a single-dimension array, you may declare its formal parameter in one of three ways: as a pointer, as a sized array, or as an unsized array. For example, to receive **i**, a function called **func1( )** can be declared as

```
void func1(int *x) /* pointer */
{
```

```
    .
    .
    .
}
```

or

```
void func1(int x[10]) /* sized array */
{
     .
     .
     .
}
```

or finally as

```
void func1(int x[]) /* unsized array */
{
     .
     .
     .
}
```

All three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. The first declaration actually uses a pointer. The second employs the standard array declaration. In the final version, a modified version of an array declaration simply specifies that an array of type **int** of some length is to be received. As you can see, the length of the array doesn't matter as far as the function is concerned because C/C++ performs no bounds checking. In fact, as far as the compiler is concerned,

```
void func1(int x[32])
{
    .
    .
    .
}
```

also works because the compiler generates code that instructs **func1( )** to receive a pointer—it does not actually create a 32-element array.

Null-Terminated Strings

By far the most common use of the one-dimensional array is as a character string. C++ supports two types of strings. The first is the *null-terminated string*, which is a null-terminated character array. (A null is zero.) Thus a null-terminated string contains the characters that comprise the string followed by a null. This is the only type of string defined by C, and it is still the most widely used. Sometimes null-terminated strings are called *C-strings*. C++ also defines a string class, called **string**, which provides an object-oriented approach to string handling. It is described later in this book. Here, null-terminated strings are examined.

When declaring a character array that will hold a null-terminated string, you need to declare it to be one character longer than the largest string that it is to hold. For example, to declare an array **str** that can hold a 10-character string, you would write

```
char str[11];
```

This makes room for the null at the end of the string.

When you use a quoted string constant in your program, you are also creating a null-terminated string. A *string constant* is a list of characters enclosed in double quotes. For example,

```
"hello there"
```

You do not need to add the null to the end of string constants manually—the compiler does this for you automatically.

C/C++ supports a wide range of functions that manipulate null-terminated strings. The most common are

Name	Function
strcpy(*s1*, *s2*)	Copies *s2* into *s1*.
strcat(*s1*, *s2*)	Concatenates *s2* onto the end of *s1*.
strlen(*s1*)	Returns the length of *s1*.
strcmp(*s1*, *s2*)	Returns 0 if *s1* and *s2* are the same; less than 0 if *s1*<*s2*; greater than 0 if *s1*>*s2*.
strchr(*s1*, *ch*)	Returns a pointer to the first occurrence of *ch* in *s1*.
strstr(*s1*, *s2*)	Returns a pointer to the first occurrence of *s2* in *s1*.

These functions use the standard header file **string.h**. (C++ programs can also use the C++-style header **<cstring>**.) The following program illustrates the use of these string functions:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  printf("lengths: %d %d\n", strlen(s1), strlen(s2));

  if(!strcmp(s1, s2)) printf("The strings are equal\n");

  strcat(s1, s2);
  printf("%s\n", s1);

  strcpy(s1, "This is a test.\n");
  printf(s1);
  if(strchr("hello", 'e')) printf("e is in hello\n");
  if(strstr("hi there", "hi")) printf("found hi");

  return 0;
}
```

If you run this program and enter the strings "**hello**" and "**hello**", the output is

```
lengths: 5 5
The strings are equal
hellohello
This is a test.
e is in hello
found hi
```

Remember, **strcmp( )** returns false if the strings are equal. Be sure to use the logical operator **!** to reverse the condition, as just shown, if you are testing for equality.

Although C++ defines a string class, null-terminated strings are still widely used in existing programs. They will probably stay in wide use because they offer a high level of efficiency and afford the programmer detailed control of string operations. However, for many simple string-handling chores, C++'s string class provides a convenient alternative.

Two-Dimensional Arrays

C/C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, essentially, an array of one-dimensional arrays. To declare a two-dimensional integer array **d** of size 10,20, you would write

```
int d[10][20];
```

Pay careful attention to the declaration. Some other computer languages use commas to separate the array dimensions; C/C++, in contrast, places each dimension in its own set of brackets.

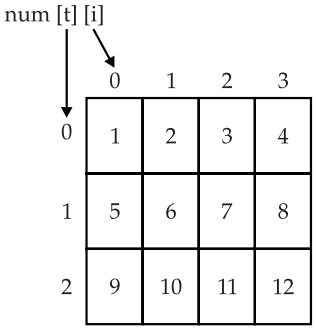
Similarly, to access point 1,2 of array **d**, you would use

```
d[1][2]
```

The following example loads a two-dimensional array with the numbers 1 through 12 and prints them row by row.

```
#include <stdio.h>

int main(void)
{
  int t, i, num[3][4];

  for(t=0; t<3; ++t)
    for(i=0; i<4; ++i)
      num[t][i] = (t*4)+i+1;

  /* now print them out */
  for(t=0; t<3; ++t) {
    for(i=0; i<4; ++i)
      printf("%3d ", num[t][i]);
    printf("\n");
  }

  return 0;
}
```

In this example, **num[0][0]** has the value 1, **num[0][1]** the value 2, **num[0][2]** the value 3, and so on. The value of **num[2][3]** will be 12. You can visualize the **num** array as shown here:



Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second indicates the column. This means that the rightmost index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory. See Figure 4-2 for a graphic representation of a two-dimensional array in memory.

In the case of a two-dimensional array, the following formula yields the number of bytes of memory needed to hold it:

bytes = size of 1st index x size of 2nd index x sizeof(base type)

Therefore, assuming 4-byte integers, an integer array with dimensions 10,5 would have

10 x 5 x 4

or 200 bytes allocated.

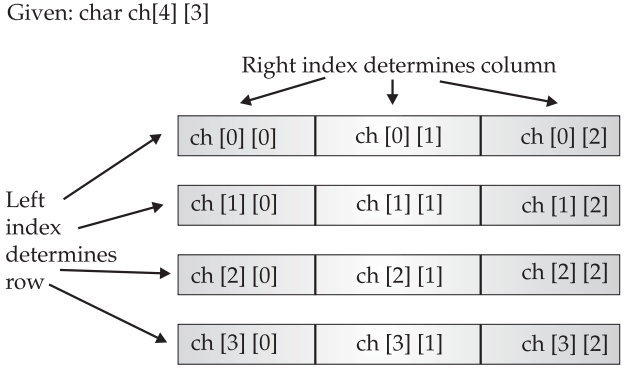


Figure 4-2. *A two-dimensional array in memory*

When a two-dimensional array is used as an argument to a function, only a pointer to the first element is actually passed. However, the parameter receiving a two-dimensional array must define at least the size of the rightmost dimension. (You can specify the left dimension if you like, but it is not necessary.) The rightmost dimension is needed because the compiler must know the length of each row if it is to index the array correctly. For example, a function that receives a two-dimensional integer array with dimensions 10,10 is declared like this:

```
void func1(int x[][10])
{
     .
     .
     .
}
```

The compiler needs to know the size of the right dimension in order to correctly execute expressions such as

```
x[2][4]
```

inside the function. If the length of the rows is not known, the compiler cannot determine where the third row begins.

The following short program uses a two-dimensional array to store the numeric grade for each student in a teacher's classes. The program assumes that the teacher has three classes and a maximum of 30 students per class. Notice the way the array **grade** is accessed by each of the functions.

```
/* A simple student grades database. */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define CLASSES  3
#define GRADES  30

int grade[CLASSES][GRADES];

void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);
```

```
int main(void)
{
  char ch, str[80];

  for(;;) {
    do {
      printf("(E)nter grades\n");
      printf("(R)eport grades\n");
      printf("(Q)uit\n");
      gets(str);
      ch = toupper(*str);
    } while(ch!='E' && ch!='R' && ch!='Q');

    switch(ch) {
      case 'E':
        enter_grades();
        break;
      case 'R':
        disp_grades(grade);
        break;
      case 'Q':
        exit(0);
    }
  }

  return 0;
}

/* Enter the student's grades. */
void enter_grades(void)
{
  int t, i;

  for(t=0; t<CLASSES; t++) {
    printf("Class # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      grade[t][i] = get_grade(i);
  }
}

/* Read a grade. */
int get_grade(int num)
```

```
{
  char s[80];

  printf("Enter grade for student # %d:\n", num+1);
  gets(s);
  return(atoi(s));
}

/* Display grades. */
void disp_grades(int g[][GRADES])
{
  int t, i;

  for(t=0; t<CLASSES; ++t) {
    printf("Class # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      printf("Student #%d is %d\n", i+1, g[t][i]);
  }
}
```

Arrays of Strings

It is not uncommon in programming to use an array of strings. For example, the input processor to a database may verify user commands against an array of valid commands. To create an array of null-terminated strings, use a two-dimensional character array. The size of the left index determines the number of strings and the size of the right index specifies the maximum length of each string. The following code declares an array of 30 strings, each with a maximum length of 79 characters, plus the null terminator.

```
char str_array[30][80];
```

It is easy to access an individual string: You simply specify only the left index. For example, the following statement calls **gets( )** with the third string in **str_array**.

```
gets(str_array[2]);
```

The preceding statement is functionally equivalent to

```
gets(&str_array[2][0]);
```

but the first of the two forms is much more common in professionally written C/C++ code.

To better understand how string arrays work, study the following short program, which uses a string array as the basis for a very simple text editor:

```
/* A very simple text editor. */
#include <stdio.h>

#define MAX 100
#define LEN 80

char text[MAX][LEN];

int main(void)
{
  register int t, i, j;

  printf("Enter an empty line to quit.\n");

  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break; /* quit on blank line */
  }

  for(i=0; i<t; i++) {
    for(j=0; text[i][j]; j++) putchar(text[i][j]);
    putchar('\n');
  }

  return 0;
}
```

This program inputs lines of text until a blank line is entered. Then it redisplays each line one character at a time.

Multidimensional Arrays

C/C++ allows arrays of more than two dimensions. The exact limit, if any, is determined by your compiler. The general form of a multidimensional array declaration is

type name[*Size1*][*Size2*][*Size3*]. . .[*SizeN*];

Arrays of more than three dimensions are not often used because of the amount of memory they require. For example, a four-dimensional character array with dimensions 10,6,9,4 requires

```
10 * 6 * 9 * 4
```

or 2,160 bytes. If the array held 2-byte integers, 4,320 bytes would be needed. If the array held **double**s (assuming 8 bytes per **double**), 17,280 bytes would be required. The storage required increases exponentially with the number of dimensions. For example, if a fifth dimension of size 10 was added to the preceding array, then 172,800 bytes would be required.

In multidimensional arrays, it takes the computer time to compute each index. This means that accessing an element in a multidimensional array can be slower than accessing an element in a single-dimension array.

When passing multidimensional arrays into functions, you must declare all but the leftmost dimension. For example, if you declare array **m** as

```
int m[4][3][6][5];
```

a function, **func1( )**, that receives **m**, would look like this:

```
void func1(int d[][3][6][5])
{
  .
  .
  .
}
```

Of course, you can include the first dimension if you like.

Indexing Pointers

In C/C++, pointers and arrays are closely related. As you know, an array name without an index is a pointer to the first element in the array. For example, consider the following array.

```
char p[10];
```

The following statements are identical:

```
p
&p[0]
```

Put another way,

```
p == &p[0]
```

evaluates to true because the address of the first element of an array is the same as the address of the array.

As stated, an array name without an index generates a pointer. Conversely, a pointer can be indexed as if it were declared to be an array. For example, consider this program fragment:

```
int *p, i[10];
p = i;
p[5] = 100;  /* assign using index */
*(p+5) = 100; /* assign using pointer arithmetic */
```

Both assignment statements place the value 100 in the sixth element of **i**. The first statement indexes **p**; the second uses pointer arithmetic. Either way, the result is the same. (Chapter 5 discusses pointers and pointer arithmetic.)

This same concept also applies to arrays of two or more dimensions. For example, assuming that **a** is a 10-by-10 integer array, these two statements are equivalent:

```
a
&a[0][0]
```

Furthermore, the 0,4 element of **a** may be referenced two ways: either by array indexing, **a[0][4]**, or by the pointer, ***((int *)a+4)**. Similarly, element 1,2 is either **a[1][2]** or ***((int *)a+12)**. In general, for any two-dimensional array

a[j][k] is equivalent to *((*base-type* *)a+(j**row length*)+k)

The cast of the pointer to the array into a pointer of its base type is necessary in order for the pointer arithmetic to operate properly. Pointers are sometimes used to access arrays because pointer arithmetic is often faster than array indexing.

A two-dimensional array can be reduced to a pointer to an array of one-dimensional arrays. Therefore, using a separate pointer variable is one easy way to use pointers to access elements within a row of a two-dimensional array. The following function illustrates this technique. It will print the contents of the specified row for the global integer array **num**:

```
int num[10][10];
  .
```

```
  .
  .
void  pr_row(int j)
{
  int *p, t;

  p = (int *) &num[j][0]; /* get address of first
                             element in row j */

  for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

You can generalize this routine by making the calling arguments be the row, the row length, and a pointer to the first array element, as shown here:

```
void pr_row(int j, int row_dimension, int *p)
{
  int t;

  p = p + (j * row_dimension);

  for(t=0; t<row_dimension; ++t)
    printf("%d ", *(p+t));
}
  .
  .
  .
void f(void)
{
  int num[10][10];

  pr_row(0, 10, (int *) num); /* print first row */
}
```

Arrays of greater than two dimensions may be reduced in a similar way. For example, a three-dimensional array can be reduced to a pointer to a two-dimensional array, which can be reduced to a pointer to a single-dimension array. Generally, an *n*-dimensional array can be reduced to a pointer and an (*n*-1)-dimensional array. This new array can be reduced again with the same method. The process ends when a single-dimension array is produced.

Array Initialization

C/C++ allows the initialization of arrays at the time of their declaration. The general form of array initialization is similar to that of other variables, as shown here:

type_specifier array_name[*size1*]. . .[*sizeN*] = { *value_list* };

The *value_list* is a comma-separated list of values whose type is compatible with *type_specifier*. The first value is placed in the first position of the array, the second value in the second position, and so on. Note that a semicolon follows the }.

In the following example, a 10-element integer array is initialized with the numbers 1 through 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

This means that **i[0]** will have the value 1 and **i[9]** will have the value 10.

Character arrays that hold strings allow a shorthand initialization that takes the form:

char *array_name*[*size*] = "*string*";

For example, this code fragment initializes **str** to the phrase "I like C++".

```
char str[11] = "I like C++";
```

This is the same as writing

```
char str[11] = {'I', ' ', 'l', 'i', 'k', 'e',' ', 'C',
                '+', '+', '\0'};
```

Because null-terminated strings end with a null, you must make sure that the array you declare is long enough to include the null. This is why **str** is 11 characters long even though "I like C++" is only 10. When you use the string constant, the compiler automatically supplies the null terminator.

Multidimensional arrays are initialized the same as single-dimension ones. For example, the following initializes **sqrs** with the numbers 1 through 10 and their squares.

```
int sqrs[10][2] = {
  1, 1,
```

```
  2, 4,
  3, 9,
  4, 16,
  5, 25,
  6, 36,
  7, 49,
  8, 64,
  9, 81,
  10, 100
};
```

When initializing a multidimensional array, you may add braces around the initializers for each dimension. This is called *subaggregate grouping*. For example, here is another way to write the preceding declaration.

```
int sqrs[10][2] = {
  {1, 1},
  {2, 4},
  {3, 9},
  {4, 16},
  {5, 25},
  {6, 36},
  {7, 49},
  {8, 64},
  {9, 81},
  {10, 100}
};
```

When using subaggregate grouping, if you don't supply enough initializers for a given group, the remaining members will be set to zero automatically.

Unsized Array Initializations

Imagine that you are using array initialization to build a table of error messages, as shown here:

```
char e1[12] = "Read error\n";
char e2[13] = "Write error\n";
char e3[18] = "Cannot open file\n";
```

As you might guess, it is tedious to count the characters in each message manually to determine the correct array dimension. Fortunately, you can let the compiler

automatically calculate the dimensions of the arrays. If, in an array initialization statement, the size of the array is not specified, the C/C++ compiler automatically creates an array big enough to hold all the initializers present. This is called an *unsized array*. Using this approach, the message table becomes

```
char e1[] = "Read error\n";
char e2[] = "Write error\n";
char e3[] = "Cannot open file\n";
```

Given these initializations, this statement

```
printf("%s has length %d\n",  e2,  sizeof e2);
```

will print

```
Write error has length 13
```

Besides being less tedious, unsized array initialization allows you to change any of the messages without fear of using incorrect array dimensions.

Unsized array initializations are not restricted to one-dimensional arrays. For multidimensional arrays, you must specify all but the leftmost dimension. (The other dimensions are needed to allow the compiler to index the array properly.) In this way, you may build tables of varying lengths and the compiler automatically allocates enough storage for them. For example, the declaration of **sqrs** as an unsized array is shown here:

```
int sqrs[][2] = {
  {1, 1},
  {2, 4},
  {3, 9},
  {4, 16},
  {5, 25},
  {6, 36},
  {7, 49},
  {8, 64},
  {9, 81},
  {10, 100}
};
```

The advantage of this declaration over the sized version is that you may lengthen or shorten the table without changing the array dimensions.

A Tic-Tac-Toe Example

The longer example that follows illustrates many of the ways that you can manipulate arrays with C/C++. This section develops a simple tic-tac-toe program. Two-dimensional arrays are commonly used to simulate board game matrices.

The computer plays a very simple game. When it is the computer's turn, it uses **get_computer_move( )** to scan the matrix, looking for an unoccupied cell. When it finds one, it puts an **O** there. If it cannot find an empty location, it reports a draw game and exits. The **get_player_move( )** function asks you where you want to place an **X.** The upper-left corner is location 1,1; the lower-right corner is 3,3.

The matrix array is initialized to contain spaces. Each move made by the player or the computer changes a space into either an X or an O. This makes it easy to display the matrix on the screen.

Each time a move has been made, the program calls the **check( )** function. This function returns a space if there is no winner yet, an X if you have won, or an O if the computer has won. It scans the rows, the columns, and then the diagonals, looking for one that contains either all X's or all O's.

The **disp_matrix( )** function displays the current state of the game. Notice how initializing the matrix with spaces simplified this function.

The routines in this example all access the **matrix** array differently. Study them to make sure that you understand each array operation.

```
/* A simple Tic Tac Toe game. */
#include <stdio.h>
#include <stdlib.h>

char matrix[3][3];  /* the tic tac toe matrix */

char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void);

int main(void)
{
  char done;

  printf("This is the game of Tic Tac Toe.\n");
  printf("You will be playing against the computer.\n");

  done =  ' ';
```

```
  init_matrix();
  do{
    disp_matrix();
    get_player_move();
    done = check(); /* see if winner */
    if(done!= ' ') break; /* winner!*/
    get_computer_move();
    done = check(); /* see if winner */
  } while(done== ' ');
  if(done=='X') printf("You won!\n");
  else printf("I won!!!!\n");
  disp_matrix(); /* show final positions */

  return 0;
}

/* Initialize the matrix. */
void init_matrix(void)
{
  int i, j;

  for(i=0; i<3; i++)
    for(j=0; j<3; j++) matrix[i][j] =  ' ';
}

/* Get a player's move. */
void get_player_move(void)
{
  int x, y;

  printf("Enter X,Y coordinates for your move: ");
  scanf("%d%*c%d", &x, &y);

  x--; y--;

  if(matrix[x][y]!= ' '){
    printf("Invalid move, try again.\n");
    get_player_move();
  }
  else matrix[x][y] = 'X';
}
```

```
/* Get a move from the computer. */
void get_computer_move(void)
{
  int i, j;
  for(i=0; i<3; i++){
    for(j=0; j<3; j++)
      if(matrix[i][j]==' ') break;
    if(matrix[i][j]==' ') break;
  }

  if(i*j==9)  {
    printf("draw\n");
    exit(0);
  }
  else
    matrix[i][j] = 'O';
}

/* Display the matrix on the screen. */
void disp_matrix(void)
{
  int t;

  for(t=0; t<3; t++) {
    printf(" %c | %c | %c ",matrix[t][0],
            matrix[t][1], matrix [t][2]);
    if(t!=2) printf("\n---|---|---\n");
  }
  printf("\n");
}

/* See if there is a winner. */
char check(void)
{
  int i;

  for(i=0; i<3; i++)  /* check rows */
    if(matrix[i][0]==matrix[i][1] &&
       matrix[i][0]==matrix[i][2]) return matrix[i][0];

  for(i=0; i<3; i++)  /* check columns */
    if(matrix[0][i]==matrix[1][i] &&
```

```
      matrix[0][i]==matrix[2][i]) return matrix[0][i];

  /* test diagonals */
  if(matrix[0][0]==matrix[1][1] &&
     matrix[1][1]==matrix[2][2])
       return matrix[0][0];

  if(matrix[0][2]==matrix[1][1] &&
     matrix[1][1]==matrix[2][0])
       return matrix[0][2];

  return ' ';
}
```